ABSTRACT

In unit testing, a program is decomposed into units which are collections of functions. A part of unit can be tested by generating inputs for a single entry function. The entry function may contain pointer arguments, in which case the inputs to the unit are memory graphs. The paper addresses the problem of automating unit testing with memory graphs as inputs. The approach used builds on previous work combining symbolic and concrete execution, and more specifically, using such a combination to generate test inputs to explore all feasible execution paths. The current work develops a method to represent and track constraints that capture the behavior of a symbolic execution of a unit with memory graphs as inputs. Moreover, an efficient constraint solver is proposed to facilitate incremental generation of each test inputs. Finally, CUTE, a tool implementing the method is described together with the results of applying CUTE to real-world examples of C code.

Categories and Subject Descriptions: D.2.5 [Software Engineering]: Testing and Debugging

General Terms: Reliability/Verification

Keywords: concolic testing, random testing, explicit path model-checking, data structure testing, unit testing, testing C programs.

1. INTRODUCTION

Unit testing is a method for modular testing of a program’s functional behavior. A program is decomposed into units, where each unit is a collection of functions, and the units are independently tested. Such testing requires specification of values for the inputs (or test inputs) to the unit. Manual specification of such values is labor intensive and cannot guarantee that all possible behaviors of the unit will be observed during the testing.

In order to improve the range of behaviors observed (or test coverage), several techniques have been proposed to automatically generate values for the inputs. One such technique is to randomly choose the values over the domain of potential inputs [4, 8, 10, 21]. The problem with such random testing is two fold: first, many sets of values may lead to the same observable behavior and are thus redundant, and second, the probability of selecting particular inputs that cause buggy behavior may be astronomically small [20].

One approach which addresses the problem of redundant executions and increases test coverage is symbolic execution [1, 3, 9, 22, 24, 25, 26, 29]. In symbolic execution, a program is executed using symbolic variables in place of concrete values for inputs. Each conditional expression in the program represents a constraint that determines an execution path. Observe that the feasible executions of a program can be represented as a tree, where the branch points in a program are internal nodes of the tree. The goal is to generate concrete values for inputs which would result in different paths being taken. The classic approach is to use depth first exploration of the paths by backtracking [24]. Unfortunately, for large or complex units, it is computationally intractable to precisely maintain and solve the constraints required for test generation.

To the best of our knowledge, Larran and Austin were the first to propose combining concrete and symbolic execution [16]. In their approach, the program is executed on some user-provided concrete input values. Symbolic path constraints are generated for the specific execution. These constraints are solved, if feasible, to see whether there are potential input values that would have led to a violation along the same execution path. This improves coverage while avoiding the computational cost associated with full-blown symbolic execution which exercises all possible execution paths.

Godfrey et al. proposed incrementally generating test inputs by combining concrete and symbolic execution [11]. In Godfrey et al.’s approach, during a concrete execution, a composition of symbolic constraints along the path of the execution is generated. These constraints are modified and then solved, if feasible, to generate further test inputs which would direct the program along alternative paths. Specifically, they systematically negate the conjuncts in the path’s constraint to provide a depth first exploration of all paths in the computation tree. If it is not feasible to solve the modified constraints, Godfrey et al. propose simply substituting random concrete values.

A challenge in applying Godfrey et al.’s approach is to provide mechanisms which extract and solve the constraints generated by a program. This problem is particularly com-
Programs Have Bugs
Why Program Testing?

- Programmer familiarity
- Concrete input for debugging
- No false positives
- Easy regression
Why Automated Testing?
Automated Testing Hits the Mainstream
Automated Testing Hits the Mainstream

Generate unit tests for your code with IntelliTest

10/04/2015 • 5 minutes to read • Contributors: ●●●●●●●● all

In this article

Availability and extensions
   Explore: Use IntelliTest to explore your code and generate unit tests
   Persist: Save the unit tests as a regression suite
   Assist: Use IntelliTest to focus code exploration
   Specify: Use IntelliTest to validate correctness properties that you specify in code

Q & A

IntelliTest explores your .NET code to generate test data and a suite of unit tests. For every statement in the code, a test input is generated that will execute that statement. A case analysis is performed for every conditional
Automated Testing Hits the Mainstream

Generate unit tests for your code with IntelliTest

10/04/2015 • 5 minutes to read • Contributors

In this article

Availability and extensions

Explore: Use IntelliTest to explore your code and
Persist: Save the unit tests as a regression suite
Assist: Use IntelliTest to focus code exploration
Specify: Use IntelliTest to validate correctness pro
Q & A

IntelliTest explores your .NET code to generate tests. For every statement in the code, a test executes that statement. A case analysis is performed.

Finding BIOS Vulnerabilities with Symbolic Execution and Virtual Platforms

By Engblom, Jakob (Intel), published on June 8, 2017

Finding vulnerabilities in code is part of the constant security game between attackers and defenders. An attacker only needs to find one opening to be successful, while a defender needs to search for and plug all or at least most of the holes in a system. Thus, a defender needs more effective tools than the attacker to come out ahead.
Automated Testing Hits the Mainstream

Generate unit tests for your code with IntelliJ

10/04/2015 • 5 minutes to read • Contributions

In this article

Availability and extensions
Explore: Use IntelliJTest to explore your code and Persist: Save the unit tests as a regression suite Assist: Use IntelliJTest to focus code exploration Specify: Use IntelliJTest to validate correctness properties

IntelliTest explores your .NET code to generate tests. For every statement in the code, a test is created to execute that statement. A case analysis is performed on the input data ranges and execution paths of the code, and the test is constructed accordingly.
Automated Test Generation Trend

- 1976: King’76, Clarke’76, Howden’77
- 2000: Java PathFinder
- **2001**: Started my PhD UIUC
- 2001: SLAM/Blast: Automatic predicate abstraction
- **2001**: Java PathExplorer: Runtime Verification
- **2003**: Runtime monitoring with Eagle (Internship)
- 2003: Generalized Symbolic Execution
Automated Test Generation Trend

- 1976: King’76, Clarke’76, Howden’77
- 2000: Java PathFinder
- **2001**: Started my PhD UIUC
- **2001**: SLAM/Blast: Automatic predicate abstraction
- **2001**: Java PathExplorer: Runtime Verification
- **2003**: Runtime monitoring with Eagle (Internship)
- **2003**: Generalized Symbolic Execution
- **2005**: DART: Directed Automated Random Testing (Internship)
- **2005**: CUTE: A Concolic Unit Testing Engine for C
- **2006**: jCUTE: Concolic Testing for Multi-threaded programs

Symbolic JPF, KLEE, CREST, S²E, Angr, Veritesting, Mayhem, Triton, Jalangi, CATG
In concolic testing, what does “concrete execution” mean?

I came across the terms "concrete & symbolic execution" when I was going through the concept of concolic testing. (The article mentioned there, "CUTE: A concolic unit testing engine for C", uses that term in its abstract section.)

"The approach used builds on previous work combining symbolic and concrete execution, and more specifically, using such a combination to generate test inputs to explore all feasible execution paths."

Can anyone please confirm what "concrete execution" means? In spite of my search, I could not find any direct citations / explicit statements.

From what I have understood, "concrete execution" means "the execution of a program with actual input values unlike symbolic execution, which assumes symbolic values to variables, inputs etc."

If I am wrong, please correct me (if possible with a small example).

Concolic execution is a mix between CONCrete execution and symbOLIC execution, with the purpose of feasibility.

Symbolic execution allows us to execute a program through all possible execution paths, thus achieving all possible path conditions (path condition = the set of logical constraints that takes us to a...
What is Concolic testing?

• Combine \textit{concrete execution} and \textit{symbolic execution}

\textbf{Concrete + Symbolic} = \textbf{Concolic}
Concolic testing (a portmanteau of concrete and symbolic) is a hybrid software verification technique that performs symbolic execution, a classical technique that treats program variables as symbolic variables, along a concrete execution (testing on particular inputs) path.

Concolic testing - Wikipedia
https://en.wikipedia.org/wiki/Concolic_testing

Concolic testing (a portmanteau of concrete and symbolic) is a hybrid software verification technique that performs symbolic execution, a classical technique that treats program variables as symbolic variables, along a concrete execution (testing on particular inputs) path.

Birth of concolic testing · Example · Algorithm · Commercial success

Concolic Fuzzing - Generating Software Tests - Fuzzing Book
https://www.fuzzingbook.org/html/ConcolicFuzzer

The idea of concolic execution over a function is as follows: We start with a sample input for the function, and execute the function under trace. At each point the ...

[PDF] Symbolic and Concolic Execution - Verimag
www-verimag.imag.fr/~mounier/Enseignement/Software_Security

Each theory comes with a set of axioms (FOL formulas), called A, which only contain elements from the signature. The predicates and functions in have no ...

why on earth is concolic execution better? · Issue #907 · klee/kle...
Goal

• Automated Unit Testing of real-world C and Java Programs
  • Generate test inputs
  • Execute unit under test on generated test inputs
    • so that all reachable statements are executed
  • Any assertion violation gets caught
Goal

• Automated Unit Testing of real-world C and Java Programs
  • Generate test inputs
  • Execute unit under test on generated test inputs
    • so that all reachable statements are executed
  • Any assertion violation gets caught

• Concolic Testing Approach:
  • Explore all execution paths of an unit for all possible inputs
Computation Tree

• Can be seen as a binary tree with possibly infinite depth
  - Computation tree
• Each node represents the execution of a “if then else” statement
• Each edge represents the execution of a sequence of non-conditional statements
• Each path in the tree represents an equivalence class of inputs
Concolic Testing Approach

int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}

- Random Test Driver:
  - random values for x and y
- Probability of reaching ERROR is extremely low
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
Concolic Testing Approach

```c
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Concrete Execution

<table>
<thead>
<tr>
<th>concrete state</th>
<th>symbolic state</th>
<th>path condition</th>
</tr>
</thead>
</table>
| ```
| x = 22, y = 7, z = 14 |
| x = x_0, y = y_0, z = 2*y_0 |
|``` |
Concolic Testing Approach

```c
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Concrete Execution:
- Concrete state:
  - x = 22, y = 7, z = 14

Symbolic Execution:
- Symbolic state:
  - x = x₀, y = y₀, z = 2*y₀
- Path condition:
  - 2*y₀ != x₀
Concolic Testing Approach

```c
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Concrete Execution

Symbolic Execution

Concrete state: solve $2y_0 = x_0$

Solution: $x_0 = 2$, $y_0 = 1$

Path condition: $2y_0 \neq x_0$

Concrete state:

- $x = 22$, $y = 7$, $z = 14$

Symbolic state:

- $x = x_0$, $y = y_0$, $z = 2y_0$
Concolic Testing Approach

```c
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

<table>
<thead>
<tr>
<th>Concrete Execution</th>
<th>Symbolic Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>concrete state</strong></td>
<td><strong>symbolic state</strong></td>
</tr>
<tr>
<td>x = 2, y = 1</td>
<td>x = x₀, y = y₀</td>
</tr>
<tr>
<td><strong>path condition</strong></td>
<td></td>
</tr>
</tbody>
</table>
Concolic Testing Approach

```c
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Concrete Execution

Symbolic Execution

**Concrete state**
- \(x = 2, \ y = 1, \ z = 2\)

**Symbolic state**
- \(x = x_0, \ y = y_0, \ z = 2y_0\)

**Path condition**
- \(x > y + 10\)
Concolic Testing Approach

```c
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Concrete Execution
- `x = 2, y = 1, z = 2`
- `x = x_0, y = y_0, z = 2*y_0`

Symbolic Execution
- `2*y_0 == x_0`
Concolic Testing Approach

```c
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Concrete Execution

<table>
<thead>
<tr>
<th>Concrete state</th>
<th>Symbolic state</th>
<th>Path condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 2, y = 1,</td>
<td>x = x₀, y = y₀,</td>
<td>2*y₀ == x₀</td>
</tr>
<tr>
<td>z = 2</td>
<td>z = 2*y₀</td>
<td>x₀ ≤ y₀+10</td>
</tr>
</tbody>
</table>
Concolic Testing Approach

int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}

Concrete Execution

Symbolic Execution

Solve: (2*y₀ == x₀) ∧ (x₀ > y₀ + 10)
Solution: x₀ = 30, y₀ = 15

2*y₀ == x₀
x₀ > y₀+10

x = 2, y = 1,  
z = 2
x = x₀, y = y₀,  
z = 2*y₀
Concolic Testing Approach

```c
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Concrete Execution

- **Concrete State**: $x = 30$, $y = 15$
- **Path Condition**: $x = x_0$, $y = y_0$

Symbolic Execution

- **Symbolic State**: $x = x_0$, $y = y_0$
Concolic Testing Approach

```c
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Concrete Execution

Symbolic Execution

**Concrete State**: x = 30, y = 15

**Symbolic State**: x = x₀, y = y₀

**Path Condition**: 2*y₀ == x₀, x₀ > y₀ + 10

**Program Error**
Explicit Path (not State) Model Checking

- Traverse all execution paths one by one to detect errors
  - assertion violations
  - program crash
  - uncaught exceptions

- combine with address sanitizer to discover memory errors
Explicit Path (not State) Model Checking

- Traverse all execution paths one by one to detect errors
  - assertion violations
  - program crash
  - uncaught exceptions

- Combine with address sanitizer to discover memory errors
Explicit Path (not State) Model Checking

- Traverse all execution paths one by one to detect errors
  - assertion violations
  - program crash
  - uncaught exceptions
- Combine with address sanitizer to discover memory errors
Explicit Path (not State) Model Checking

- Traverse all execution paths one by one to detect errors
  - assertion violations
  - program crash
  - uncaught exceptions

- combine with address sanitizer to discover memory errors
Explicit Path (not State) Model Checking

- Traverse all execution paths one by one to detect errors
  - assertion violations
  - program crash
  - uncaught exceptions

- Combine with address sanitizer to discover memory errors
Explicit Path (not State) Model Checking

- Traverse all execution paths one by one to detect errors
  - assertion violations
  - program crash
  - uncaught exceptions

- Combine with address sanitizer to discover memory errors
Novelty: Simultaneous Concrete and Symbolic Execution

```c
int foo (int v) {
    return (v*v) % 50;
}

void testme (int x, int y) {
    z = foo (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Concrete Execution

- Concrete state: $x = 22$, $y = 7$

Symbolic Execution

- Symbolic state: $x = x_0$, $y = y_0$

Path condition

- $x > y+10$
Novelty: Simultaneous Concrete and Symbolic Execution

```c
int foo (int v) {
    return (v*v) % 50;
}

void testme (int x, int y) {
    z = foo (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Concrete Execution

Symbolic Execution

Concrete state

Symbolic state

Path condition

Solve: \((y_0 \cdot y_0) \% 50 = x_0\)

Don’t know how to solve!

Stuck?

\(x = 22\), \(y = 7\), \(z = 49\)

\(x = x_0\), \(y = y_0\), \(z = (y_0 \cdot y_0) \% 50\)
Novelty: Simultaneous Concrete and Symbolic Execution

don't know how to solve!

Stuck?

Concrete Execution

Symbolic Execution

concrete state

symbolic state

path condition

void testme (int x, int y) {
    z = foo (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}

x = 22, y = 7, z = 49
x = x₀, y = y₀, z = foo (y₀)
Novelty: Simultaneous Concrete and Symbolic Execution

```c
int foo (int v) {
    return (v*v) % 50;
}

void testme (int x, int y) {
    z = foo (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Concrete Execution

Symbolic Execution

---

Solve: \((y_0 \cdot y_0) \mod 50 = x_0\)

Don’t know how to solve!

**Not Stuck!**

Use concrete state

Replace \(y_0\) by 7 (sound)

---

\(x = 22, y = 7, z = 49\)

\(x = x_0, y = y_0, z = (y_0 \cdot y_0) \mod 50\)
Novelty: Simultaneous Concrete and Symbolic Execution

```c
int foo (int v) {
    return (v*v) % 50;
}

void testme (int x, int y) {
    z = foo (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Concrete Execution

- `z = foo (y);`
- `if (z == x) {`
  - `if (x > y+10) {`
    - ERROR;
  }

Symbolic Execution

- `Solve: 49 == x_0`
- `Solution: x_0 = 49, y_0 = 7`
- `49 != x_0`

Concrete state:
- `x = 22, y = 7, z = 48`
- `x = x_0, y = y_0, z = 49`
```
int foo (int v) {
    return (v*v) % 50;
}

void testme (int x, int y) {
    z = foo (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

**Novelty: Simultaneous Concrete and Symbolic Execution**

Concrete Execution:
- \( x = 49, y = 7 \)

Symbolic Execution:
- \( x = x_0, y = y_0 \)

Path Condition:
- If \( z = x \)
- If \( x > y + 10 \)
Novelty: Simultaneous Concrete and Symbolic Execution

```c
int foo (int v) {
    return (v*v) % 50;
}

void testme (int x, int y) {
    z = foo (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Concrete Execution

Symbolic Execution

Concrete state

Symbolic state

Path condition

Program Error

x = 49, y = 7, z = 49
x = x₀, y = y₀, z = 49

2*Y₀ == X₀
X₀ > Y₀+10
Summary: Pointers and Data-Structures

Logical Input Map to symbolically represent Memory Graph pointed by an input Pointer

$$\{0 \rightarrow 1, 1 \rightarrow 236, 2 \rightarrow 1\}$$

- Pointer Constraints
  - $p \neq \text{NULL}$
  - $p = \text{NULL}$
  - $p \neq q$
  - $p = q$

- Solving Pointer Constraints
  - Construct equivalence class $[p]$ for each pointer input $p$
  - $p \neq \text{NULL}$
    - Add a node and point $[p]$ to it
  - $p = \text{NULL}$
    - Delete node pointed by $[p]$
  - $p = q$
    - Make $[p]$ and $[q]$ point to same node
  - $p \neq q$
    - Add a node and point $[p]$ or $[q]$ to it
Concolic Testing: Finding Security and Safety Bugs

Divide by 0 Error

\[ x = \frac{3}{i}; \]

Buffer Overflow

\[ a[i] = 4; \]
Concolic Testing: Finding Security and Safety Bugs

Key: Add Checks Automatically and Perform Concolic Testing

Divide by 0 Error

if (i != 0)
    x = 3 / i;
else
    ERROR;

Buffer Overflow

if (0 <= i && i < a.length)
    a[i] = 4;
else
    ERROR;
Incremental Constraint Solving

• Observation: one constraint is negated at each execution
  • $C_1 \land C_2 \land ... \land C_k$ has a satisfying assignment
  • Need to solve $C_1 \land C_2 \land ... \land \neg C_k$
  • Previous solution more or less similar to current solution
  • Eliminate non-dependent constraints
    
    $(x==1) \land (y>2) \land \neg (y==4)$

    to

    $(y>2) \land \neg (y==4)$

• Incremental Solving
  • 100 -1000 times faster than a naïve solver
Underlying Random Testing Helps

```c
foobar(int x, int y){
    if (x*x*x > 0){
        if (x>0 && y==10){
            ERROR;
        }
    } else {
        if (x>0 && y==20){
            ERROR;
        }
    }
}
```

- static analysis based model-checkers would consider both branches
  - both ERROR statements are reachable
  - false alarm

- Symbolic execution
  - gets stuck at line number 2
  - or warn that both ERRORs are reachable

- CUTE finds the only error
DART, CUTE, jCUTE, CREST, Jalangi, CATG

- DART for C
- CUTE for C and jCUTE for Java
  - 5000+ downloads (around 2010)
  - used in both academia and industry
- CREST
  - extensible open-source tool for C
- Jalangi for JavaScript Concolic Testing
- CATG for Concolic Testing of Java bytecode
  - [https://github.com/ksen007/janala2](https://github.com/ksen007/janala2)
Concolic Testing in Practice

• Led to the development of several industrial and academic automated testing and security tools
  • Projects at Intel, Google, MathWorks, NTT, SalesForce
  • PEX, SAGE, and YOGI at Microsoft
  • Apollo at IBM, and Conbol and Jalangi at Samsung
  • BitBlaze, jFuzz, Oasis, and SmartFuzz in academia
Concolic Testing in Practice

• Led to the development of several industrial and academic automated testing and security tools
  • Projects at Intel, Google, MathWorks, NTT, Salesforce
  • PEX, SAGE, and YOGI at Microsoft
  • Apollo at IBM, and Conbol and Jalangi at Samsung
  • BitBlaze, jFuzz, Oasis, and SmartFuzz in academia
Many Applications

CLOUDY, WITH A CHANCE OF EXPLOITS —
Microsoft launches “fuzzing-as-a-service” to help developers find security bugs

Project Springfield, Microsoft’s "million-dollar bug detector" now available in cloud.

SEAN GALLAGHER - 9/27/2016, 6:21 PM
Many Applications

Microsoft launches “fuzzing-as-a-service” to help developers find security bugs

Project Springfield, Microsoft’s “million-dollar bug detector” now available in cloud.

Alan Cao

For my winternship and springternship at Trail of Bits, I researched novel techniques for symbolic execution on cryptographic protocols. I analyzed various implementation-level bugs in cryptographic libraries, and built a prototype Manticore-based concolic unit testing tool, Sandshrew, that analyzed C cryptographic primitives under a symbolic and concrete environment.
Many Applications

Microsoft launches “fuzzing-as-a-service” to help developers find security bugs

Project Springfield, Microsoft’s “million-dollar bug detector” now available in cloud.

SEAN GALLAGHER - 9/27/2016, 6:21 PM

Performing Concolic Execution on Cryptographic Primitives

Alan Cao

For my winternship and springternship at Trail of Bits, I researched novel techniques for symbolic execution on cryptographic protocols. I analyzed various implementation-level bugs in cryptographic libraries, and built a prototype Manticore-based concolic unit testing tool, Sandshrew, that analyzed C cryptographic primitives under a symbolic and concrete environment.

Con2colic testing

Full Text: PDF  Get this Article

Authors: Azadeh Farzan  University of Toronto, Canada
Andreas Holzer  Vienna University of Technology, Austria
Niloofar Razavi  University of Toronto, Canada
Helmut Veith  Vienna University of Technology, Austria

Published in:
Proceeding ESEC/FSE 2013  Proceedings of the 2013 9th Joint Meeting on ESEC/FSE Pages 37-47

COMPI: Concolic Testing for MPI Applications

Publisher: IEEE

5 Author(s) Hongbo Li ; Sihuan Li ; Zachary Benavides ; Zizhong Chen ; Rajiv Gupta  View All Authors
Many Applications

Microsoft launches “fuzzing-as-a-service” to help developers find security bugs

Project Springfield, Microsoft’s “million-dollar bug detector” now available in cloud.

Sean Gallagher - 9/27/2016, 6:21 PM

Performing Concolic Execution on Cryptographic Primitives

Alan Cao

For my winternship and springternship at Trail of Bits, I researched novel techniques for symbolic execution on cryptographic protocols. I analyzed various implementation-level bugs in cryptographic libraries, and built a prototype Manticore-based concolic unit testing tool, Sandshrew, that analyzed C cryptographic primitives under a symbolic and concrete environment.

Con2colic testing

Full Text: PDF Get this Article

Authors: Azadeh Farzan University of Toronto, Canada
Andreas Holzer Vienna University of Technology, Austria
Niloofar Razavi Institutional Profile Page
Helmut Veith Vienna University of Technology, Austria

Published in:
- Proceeding ESEC/FSE 2013 Proceedings of the 2013 9th Joint Meeting on Pages 37-47

COMPI: Concolic Testing for MPI Applications

Publisher: IEEE

5 Author(s) Hongbo Li; Sihuan Li; Zachary Benavides; Zizhong Chen; Rajiv Gupta View All Authors
Many Applications

Microsoft launches “fuzzing-as-a-service” to help developers find security bugs

Project Springfield, Microsoft’s "million-dollar bug detector" now available in cloud.

Alan Cao

For my winternship and springternship at Trail of Bits, I researched novel techniques for symbolic execution on cryptographic protocols. I analyzed various implementation-level bugs in cryptographic libraries, and built a prototype Manticore-based concolic unit testing tool, Sandshrew, that analyzed C cryptographic primitives under a symbolic and concrete environment.

Con2colic testing

Authors: Azadeh Farzan, Andreas Holzer, Niloo Farzan, Helmut Veith

Published in: ESEC/FSE 2013

COMPI: Concolic Testing for MPI Applications

Publisher: IEEE

5 Author(s) Hongbo Li, Sihuan Li, Zachary Benavides, Zizhong Chen, Rajiv Gupta

View All Authors
Many Languages

- C
- C++
- Java
- JavaScript
- Python
- PHP
- x86
- LLVM
- Java bytecode
Concolic Testing: Path-explosion Problem

Entire Computation Tree
Concolic Testing: Path-explosion Problem
Scaling Concolic Testing

- Control-flow Directed Search (CREST)
- Combining fuzzing and concolic testing (Hybrid Concolic Testing, Driller, Mayhem)
- Function Summaries (SMART, Veritesting)
- Loop Summaries (Proteus, LESE)
- State Merging using Value Summaries (MultiSE)
- Interpolation (Tracer)
- Abstract Subsumption Checking
- Pruning redundant paths (RWSet)
- Parallel techniques (Siddiqui & Khurshid, and Staats & Pasareanu)
- Incremental techniques (Person et al.)
- ...
Lessons Learned

• Focused on an important real-world problem
• Did not try to invent from the beginning
  • Tried existing approaches to solve a real problem
  • Observed limitations
  • Got insights → led to effective solutions
  • Identified novel contributions (and wrote papers)
Things We Should Have Done Differently

• If there is a big idea for a practical problem
  • Build a practical system that users can use
  • Promote the area of research
    • Your competitors are your real-friends
  • Do not hesitate to use competing techniques
    • If it helps to solve the problem
• Take feedback seriously
  • From actual users
  • And reviewers
What we do now

• We target real-world problems
• We target real software in the more popular languages
  • rather than assuming a nice clean slate for research
  • leads us to see a lot of problems
• We build prototypes before building a large system
• We release our tools as open-source software
  • so that the tools are usable by the broader community
• We release our benchmarks
# Smart Fuzzing

## Intention
- Correctness: **FairFuzz**
- Performance Bugs: **PerfFuzz**
- Custom testing Goals: **FuzzFactory**
- Semantic Fuzzing: **Semantic Fuzzing (Zest)**
- Constraint Fuzzing: **QuickSampler**

## Algorithm
- Symbolic Execution/Concolic Testing: **CUTE**
- Genetic algorithm: **AFL**
- Reinforcement Learning: **RLCheck**
- Neural Network: ???
- Bayesian Learning: ???

## Implementation
- Parallelization: LLVM, x86
- Java Virtual Machine: **JQF, RLCheck**
- Python: **RLCheck**
- RTL using FPGA: **RFuzz (Laeufer)**
Zest: Semantic Fuzzing
Padhye, Lemieux, Sen, Papadakis, Le Traon

public XMLElement genXML(Random random) {
    // Generate a random tag name
    String name = random.nextString(MAX_TAG_LENGTH);
    XMLElement node = new XMLElement(name);

    // Generate a random number of children
    int n = random.nextInt(MAX_CHILDREN);
    for (int i = 0; i < n; i++) {
        // Generate child nodes recursively
        node.addChild(genXML(random));
    }

    // Maybe insert text inside element
    if (random.nextBoolean()) {
        node.addText(random.nextString(MAX_TEXT_LENGTH));
    }
    return node;
}

- Developer writes a simple input generator as a program
- Generator restricts the space of inputs

Example generated: 
<foo><i>xyz</i><br/></foo>
Zest: New bugs discovered

- **Google Closure Compiler**: #2842, #2843, #3220, #3173
- **OpenJDK**: JDK-8190332, JDK-8190511, JDK-8190512, JDK-8190997, JDK-8191023, JDK-8191076, JDK-8191109, JDK-8191174, JDK-8191073, JDK-8193444, JDK-8193877, **CVE-2018-3214**
- **Apache Commons**: LANG-1385, COMPRESS-424, COLLECTIONS-714, **CVE-2018-11771**
- **Apache Ant**: #62655
- **Apache Maven**: #34, #57
- **Apache PDFBox**: PDFBOX-4333, PDFBOX-4338, PDFBOX-4339, **CVE-2018-8036**
- **Apache TIKA**: **CVE-2018-8017**, **CVE-2018-12418**
- **Apache BCEL**: BCEL-303, BCEL-307, BCEL-308, BCEL-309, BCEL-310, BCEL-311, BCEL-312, BCEL-313
- **Mozilla Rhino**: #405, #406, #407, #409, #410
QuickSampler, SMTSampler, GuidedSampler

Human Writes a Pre-condition on Inputs

✔ An over-approximation of valid inputs
✔ Restricts the set of inputs to be generated

Goal: sample inputs from the restricted input space

(node.left != NULL => node.val > node.left.val)
\(\land\) (node.right != NULL => node.val <= node.right.val)
Generates more **diverse** set of solutions compared to UniGen2 and SearchTreeSampler

- QuickSampler generates valid solutions
  - $10^{2.5 \pm 0.8}$ times **faster** than SearchTreeSampler
  - $10^{4.7 \pm 1.0}$ times **faster** than UniGen2
- QuickSampler generates *unique* valid solutions
  - $10^{2.3 \pm 0.7}$ times **faster** than SearchTreeSampler
  - $10^{4.4 \pm 1.1}$ times **faster** than UniGen2
Smart Fuzzing

**Intention**
- Correctness
  - FairFuzz
- Performance Bugs
  - PerfFuzz
- Custom testing Goals
  - FuzzFactory
- Semantic Fuzzing (Zest)
- Constraint Fuzzing
  - QuickSampler
  - SMTSampler (Dutra)

**Algorithm**
- Symbolic Execution/Concolic Testing
  - CUTE
- Genetic algorithm
  - AFL
- Reinforcement Learning
  - RLCheck
- Neural Network
  - ???
- Bayesian Learning
  - ???

**Implementation**
- Parallelization
  - LLVM, x86
- Java Virtual Machine
  - JQF, RLCheck
- Python
  - RLCheck
- RTL using FPGA
  - RFuzz (Laeufer)
ABSTRACT

In unit testing, a program is decomposed into units which are collections of functions. A part of unit can be tested by generating inputs for a single entry function. The entry function may contain pointer arguments, in which case the input to the function may need to be randomly chosen and then to test the function. A possible solution to this problem is to use symbolic execution, which involves randomizing the inputs to the function and then to execute the function with symbolic graphs as inputs. However, an efficient constrained solver is required to verify the incremental generation of values for the inputs. Hence, CUTE, a tool implementing the method is described together with the results of applying CUTE to real-world examples of C code.

Categories and Subject Descriptors: D.3.5 [Software Engineering]: Testing and Debugging

General Terms: Reliability, Verification

Keywords: concrete testing, random testing, explicit path world, testing, data structure testing, unit testing, testing C programs

1. INTRODUCTION

Unit testing is a method for modular testing of a pro-
gressive functional behavior. A program is decomposed into units, wherein each unit is a collection of functions, and the
function is further decomposed into a mechanism of execution of values for the inputs (or test inputs) in the unit. Manual specification of such values is labor intensive and cannot guarantee that all possible behaviors of the unit will be observed during the testing.

In order to improve the range of behaviors observed (or test coverage), several techniques have been presented to automatically generate values for the inputs. One such tech-

nique is to randomly choose the values over the domain of potential inputs [4,18,31]. The problems with such random testing in real-life test cases are: many test cases that fail to the

same input should be given a different input to avoid the

probability of selecting particular inputs that cause bugs. In addition, they are not automatically solved [20].

One approach which addresses the problem of redundant execution and increase test coverage is symbolic execu-
tion [1,5,9,13,23,25,29,30]. In symbolic execution, a pro-
gram is executed using symbolic variables in place of con-
tents of variables. Each input value is associated with a constraint that determines the execution path. A constraint represents a constraint that determines an execution path. However, the symbolic execution of a program can be represented as a tree, where the branch points are the constraints and the leaves are the input values.

To the best of our knowledge, Laxmen and Austin were the first to propose extending symbolic execution [18]. In their approach, the program is executed by assigning symbolic values to each input variable and then to execute the program. The symbolic execution is repeated until the program terminates or no further symbolic execution can be achieved.

Godfrey et al. proposed incrementally generating test inputs by combining concrete and symbolic execution [15]. In Godfrey et al.'s approach, during a concrete execution, a constraint of symbolic execution along the path of the execution is refined until a terminal constraint is reached. Only after the constraint is verified, it is then converted to a concrete value which is used to continue exploration of the program.

Specifically, they automatically refine the constraints in the path constraints to provide a depth-first exploration of all paths in the program. If it is not possible to solve the bounded constraint, Godfrey et al. propose simple substi-
tution, if feasible.

A challenge in applying Godfrey et al.'s approach is to provide methods which extract and solve the constraints encountered by a constraint. This problem is naturally open.